

Learning Latency-Aware Orchestration for Multi-Agent Systems

Xi Shi Mengxin Zheng Qian Lou
University of Central Florida
{xi320101, mengxin.zheng, qian.lou}@ucf.edu

Abstract

Multi-agent systems (MAS) coordinate multiple LLM-powered agents through structured workflows, gaining reasoning power but incurring high inference latency from multi-step execution and repeated model invocations. Existing orchestration methods primarily optimize task performance and inference cost, leaving latency largely unaddressed. In MAS, end-to-end latency is governed by the *critical execution path*, so reducing total cost alone does not reliably reduce latency. Moreover, optimizing latency while preserving accuracy remains non-trivial: naive latency optimization can misassign operator-level credit and degrade task accuracy. To address this gap, we propose **Latency-Aware Multi-agent System (LAMaS)**, a latency-aware orchestration framework for learning-based multi-agent systems. LAMaS addresses this challenge at two levels: at *training time*, it learns latency-aware execution graphs through constrained optimization with critical-path-aware credit assignment; at *inference time*, since a graph committed at training time cannot exploit runtime evidence, it complements graph construction with a lightweight controller that adaptively eliminates redundant future agent interactions as execution unfolds. Experiments on four benchmarks show that LAMaS achieves the best latency among evaluated learning-based MAS baselines, reducing end-to-end latency by over 50% while maintaining competitive or better accuracy. LAMaS is also modular and transfers to other MAS with minimal changes, consistently yielding latency reductions.

1 Introduction

Multi-Agent Systems (MAS) extend the reasoning capabilities of large language models (LLMs) by orchestrating multiple specialized agents Li et al. [2023], Wu et al. [2024], Hong et al. [2023]. However, as these systems scale to include more agents and interaction steps, inference latency becomes a key bottleneck Chen et al. [2025]. State-of-the-art agent systems already require 10–60 minutes per task in realistic settings such as computer-use Abhyankar et al. [2025] and autonomous research Kang et al. [2025], making latency an important factor for practical deployment. In MAS, this challenge is amplified because end-to-end time depends on the structure of agent interactions, not only on per-agent computation.

Recent efforts in automated multi-agent orchestration have made significant progress in optimizing task performance and inference cost [Zhang et al., 2025a, Wang et al., 2025c, Zhang et al., 2024, 2025c, Wang et al., 2025b,a, Yao et al., 2025]. Yet these methods largely treat efficiency through additive objectives such as cost, whereas latency in MAS is a structured property of the execution graph. Specifically, end-to-end latency is governed by the *critical execution path*—the longest chain of operations—rather than the total cost across all operators. As illustrated in Figure 1, two execution graphs can achieve similar accuracy and cost, yet differ substantially in latency.

However, optimizing latency while preserving accuracy in MAS is non-trivial. First, latency is non-additive over the execution graph, so a uniform latency penalty yields noisy operator-level

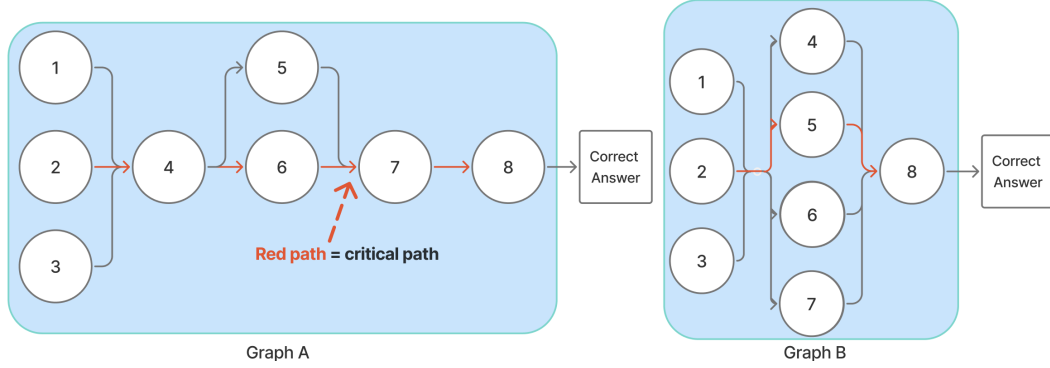


Figure 1: Two execution graphs that produce the same answer at similar cost but differ in latency. Nodes denote agent invocations and arrows denote data flow; red edges mark the critical execution path, the longest chain of dependencies that must complete before an answer is available. Graph B has a shorter critical path and therefore lower wall-clock latency than Graph A.

credit. Second, latency-driven optimization without an accuracy floor can degrade task performance unpredictably. Third, a graph fixed at training time cannot leverage execution-time evidence about whether further agent interactions remain worthwhile.

To address these challenges, we propose **Latency-Aware Multi-agent System (LAMaS)**, a latency-aware orchestration framework for multi-agent systems. At *training time*, LAMaS addresses the first two challenges through constrained optimization with critical-path-aware credit assignment, which targets the latency penalty on bottleneck operators while an explicit accuracy floor is enforced via Lagrangian relaxation. At *inference time*, it addresses the third by complementing graph construction with execution control that monitors operator completions and eliminates redundant future agent interactions.

We evaluate LAMaS on four benchmarks (GSM8K Cobbe et al. [2021], HumanEval Chen et al. [2021], MATH Hendrycks et al. [2021], and MMLU-Pro Wang et al. [2024]) against MAS baselines MaAS Zhang et al. [2025a], GDesigner Zhang et al. [2024], and AgentDropout Wang et al. [2025c]. LAMaS achieves the best latency among these baselines while maintaining competitive or better accuracy, and its mechanisms transfer to other MAS with minimal changes.

Our contributions are as follows:

- We formulate latency-aware MAS orchestration as a constrained optimization that minimizes latency under an accuracy floor, solved via Lagrangian relaxation with critical-path-aware credit assignment.
- We introduce a lightweight execution-time controller that scores partial executions under the same Lagrangian objective and supports online elimination of redundant future agent interactions.
- We demonstrate that LAMaS achieves the best latency among MAS baselines across four benchmarks while maintaining competitive or better accuracy, and show that its core mechanisms transfer to other MAS with minimal changes.

2 Related Work

LLM-Based Multi-Agent Systems. Multi-Agent Systems built on LLMs decompose complex problems into subtasks handled by specialized agents. Early frameworks such as CAMEL Li et al. [2023], AutoGen Wu et al. [2024], and MetaGPT Hong et al. [2023] demonstrated the potential of collaborative problem-solving through role-playing and structured communication. However, these approaches rely on static, hand-crafted topologies that are labor-intensive to design and difficult to adapt across tasks and difficulty levels.

Automated Multi-Agent Orchestration. To overcome the rigidity of manual designs, recent work treats orchestration as a search or optimization problem. AnyMAC Wang et al. [2025a] and SeqCV Yao et al. [2025] dynamically adjust chain length per query. Aflow Zhang et al. [2025c] and

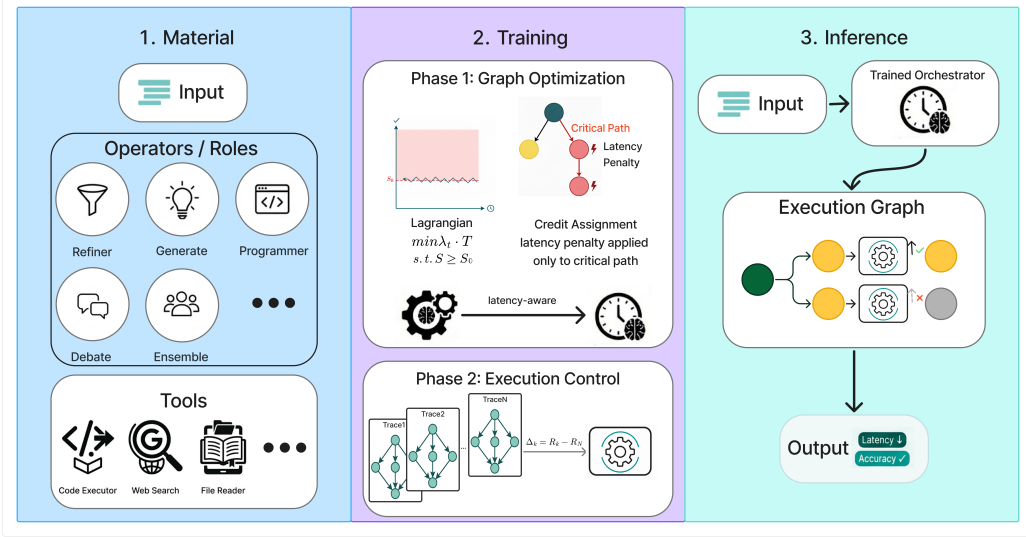


Figure 2: Overview of the LAMaS framework. **(1) Material:** A library of reusable operators and tools. **(2) Training:** Phase 1 performs latency-aware graph optimization via Lagrangian-constrained learning and critical-path credit assignment, where S_0 denotes the target accuracy floor enforced during optimization. Phase 2 trains a lightweight execution controller on the orchestrator’s traces to predict the control advantage. **(3) Inference:** The trained orchestrator first constructs the execution graph; the execution controller then performs online execution control, monitoring operator completions and adaptively eliminating redundant future agent interactions.

EvoAgentX Wang et al. [2025b] search over execution topologies using evolutionary or flow-based methods. MaAS Zhang et al. [2025a] models the search space as a probabilistic agentic supernet and trains an orchestrator via policy gradient. AgentPrune Zhang et al. [2025b] prunes redundant communication messages to obtain more economical communication topologies, while AgentDropout Wang et al. [2025c] dynamically eliminates redundant agents and communication to improve token efficiency. GDesigner Zhang et al. [2024] uses graph neural networks to architect communication topologies. These methods primarily optimize task performance, token efficiency, or inference cost. In contrast, latency is a graph-level systems objective whose bottlenecks are concentrated on the critical execution path, making it difficult to optimize with uniform additive rewards alone. Orthogonally, systems-level latency management methods such as Kairos Chen et al. [2025] optimize serving, batching, or runtime scheduling, complementary to our focus on learning latency-aware orchestration policies.

3 Methodology

3.1 Problem Setting

We consider a query-dependent multi-agent system (MAS) that dynamically composes a set of *agentic operators* to solve an input problem. An agentic operator is the basic execution unit, representing a composite agent invocation that may involve multiple LLM calls and external tool usage. Given a query x , the orchestrator constructs an execution graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, a directed acyclic graph (DAG) where each node $o \in \mathcal{V}$ is an operator invocation. The edge set $\mathcal{E}$ is induced by operator I/O flow: an edge $(o_i, o_j) \in \mathcal{E}$ exists when the output produced by o_i is used as input by o_j during execution.

We define the *critical path* as the longest path in $\mathcal{G}$ measured by execution time:

$$T = \max_{p \in \text{Paths}(\mathcal{G})} \sum_{o \in p} t(o), \quad (1)$$

where $t(o)$ denotes the execution time of operator o , and $\text{Paths}(\mathcal{G})$ denotes all source-to-sink paths in $\mathcal{G}$. The critical path determines the end-to-end latency: operators off the critical path do not affect

wall-clock time. In contrast, the total execution cost accumulates additively across all operators: $C = \sum_{o \in \mathcal{V}} c(o)$. Latency and cost therefore diverge.

Our goal is to learn an orchestrator that preserves task accuracy while minimizing end-to-end latency.

3.2 Orchestrator Architecture

LAMaS represents the orchestrator as a policy network over an *agentic supernet*—a probabilistic DAG over a fixed set of candidate operators $\mathcal{O}$ (a formulation also adopted by Zhang et al. [2025a]). The latency-aware training and credit assignment introduced below are agnostic to the specific architectural choice and apply to any orchestrator that produces a differentiable distribution over execution graphs. The orchestrator constructs $\mathcal{G}$ by sampling operators step-by-step, with the generation probability factorized as:

$$P_\theta(\mathcal{G}|x) = \prod_{\ell=1}^L \pi_\theta(\mathcal{V}_\ell \mid x, \mathcal{G}_{1:\ell-1}), \quad (2)$$

where $\mathcal{V}_\ell \subseteq \mathcal{O}$ is the operator subset selected at step ℓ . The network produces a query-aware operator-level policy via softmax over activation logits; we denote the resulting probability of operator o by $s_o := p_\theta(o \mid x, \mathcal{G}_{1:\ell-1})$. Operators are sampled from this policy sequentially without replacement until their cumulative probability first exceeds a threshold τ :

$$\mathcal{V}_\ell = \{o_1, \dots, o_k\}, \quad k = \min\left\{j : \sum_{i=1}^j s_{o_i} > \tau\right\}, \quad (3)$$

enabling the architecture to dynamically adjust width and depth per query.

3.3 Training-Time Constrained Optimization

Existing multi-agent orchestration methods typically combine accuracy and cost into a single weighted reward (e.g., $R = S - \lambda_c C$). However, as discussed in Section 1, latency and cost are structurally different objectives. Because latency is governed by the critical path, uniform penalties provide weak operator-level attribution; without an explicit accuracy floor, latency-driven optimization can degrade task performance; and a graph fixed at training time cannot leverage execution-time evidence. LAMaS addresses these via critical-path-aware credit assignment, an accuracy floor enforced by Lagrangian relaxation, and a lightweight execution-time controller. We therefore formulate MAS orchestration as a constrained optimization:

$$\min_{\theta} \mathbb{E}[\lambda_t T + \lambda_c C], \quad \text{s.t.} \quad \mathbb{E}[S] \geq S_0, \quad (4)$$

where $S \in \{0, 1\}$ is the task score, C is the execution cost in USD per query, T is the end-to-end wall-clock latency in seconds per query, and λ_t, λ_c are fixed weights controlling the relative importance of latency and cost. Here, S_0 is a fixed target accuracy floor on $\mathbb{E}[S]$, rather than a per-instance requirement, and in practice is set to the architecture’s reference accuracy (see the appendix for sensitivity to different floors).

The accuracy floor S_0 acts as a “do-no-harm” constraint: LAMaS reduces latency without falling below the reference accuracy of the underlying architecture. In our main experiments, S_0 is set to the validation accuracy of the same orchestrator trained with $\lambda_t = 0$; we analyze sensitivity to S_0 in the appendix. We solve the constrained problem via Lagrangian relaxation with an adaptive dual variable:

$$R = \lambda_S \frac{S}{S_0} - \lambda_t T - \lambda_c C, \quad (5)$$

where λ_S is the dual variable updated via projected gradient ascent:

$$\lambda_S \leftarrow \max\left(0, \lambda_S + \eta \left(1 - \frac{\bar{S}}{S_0}\right)\right), \quad (6)$$

where $\bar{S}$ denotes the batch-averaged accuracy and η is the dual learning rate. When accuracy drops below S_0 , λ_S automatically increases to prioritize correctness; when accuracy is satisfied, λ_S decreases to allow more aggressive latency reduction. Empirically, this adaptive coupling keeps accuracy fluctuating near S_0 throughout training while latency contracts steadily to its converged

value, supporting the “do-no-harm” interpretation of the constraint and removing the need to hand-tune a fixed accuracy–latency trade-off weight. Appendix C.3 visualizes this behavior on GSM8K and MMLU-Pro: latency decays steadily over training while accuracy oscillates around S_0 and λ_S adapts in response.

3.4 Critical-Path-Aware Credit Assignment

End-to-end latency is determined by the critical path. Applying the latency penalty uniformly to all operators creates a mismatch in operator-level credit assignment: operators that have little or no influence on T receive the same gradient as bottleneck operators on the critical path. We therefore use a soft bottleneck-aware weighting that preserves full penalty on the realized critical path while assigning attenuated latency credit to operators on near-critical paths.

For each operator o in the realized execution graph $\mathcal{G}$, let $\ell(o)$ denote the latency of the longest path passing through o , and define the path-criticality weight

$$w(o) = \frac{\ell(o)}{T} \in [0, 1]. \quad (7)$$

By construction $w(o) = 1$ exactly when o lies on the critical path, $w(o) \approx 1$ for operators on near-critical paths, and $w(o) \ll 1$ for operators whose longest containing path is far below T . We assign operator-level rewards as

$$R(o) = \lambda_S \frac{S}{S_0} - \lambda_t w(o) T - \lambda_c C, \quad (8)$$

so each operator receives a fraction $w(o)$ of the global latency penalty $-\lambda_t T$: critical-path operators are penalized in full, near-critical operators receive a smoothly attenuated share, and operators with negligible $w(o)$ get effectively no latency signal. This biases the gradient toward bottleneck operators while preserving softer latency attribution for near-critical paths; the values $\ell(o)$, and hence $w(o)$, are obtained alongside the critical path itself via a single forward/backward pass over $\mathcal{G}$.

3.5 Learning Objective

The orchestrator is trained with policy gradient optimization. Let ξ denote the trajectory of operator selections; the loss is

$$\mathcal{L}(\theta) = -\mathbb{E}_{\xi \sim \pi_\theta} \left[\sum_{\ell=1}^L \sum_{o \in \mathcal{V}_\ell} R(o) \log p_\theta(o \mid x, \mathcal{G}_{1:\ell-1}) \right]. \quad (9)$$

To reduce variance, we normalize rewards using exponential moving averages (EMA). Specifically, we maintain running estimates $\hat{\mu}$ and $\hat{\sigma}^2$ of the mean and variance of $R(o)$ with momentum 0.95, and normalize each reward as $(R(o) - \hat{\mu})/\hat{\sigma}$ before computing the policy gradient; this is more stable than per-batch normalization for small batch sizes.

3.6 Inference-Time Execution Control

The orchestrator commits to the full execution graph based solely on the query x , but each operator completion reveals new information—output content, emerging answer consensus, and elapsed time—unavailable at construction time. LAMaS therefore augments graph construction with an execution-time controller, implemented as a lightweight learned module. This controller can be viewed as performing an online elimination of redundant future agent interactions: at each operator completion, it evaluates whether the current partial execution should be retained or whether additional agent interactions remain worthwhile, by predicting the *control advantage*—how much better (or worse) the current partial execution would be compared to running all operators:

$$\Delta_k = R_k - R_N, \quad (10)$$

where R_k is the Lagrangian reward (Eq. 5) evaluated at the k -th operator completion and R_N is the reward under full execution. $\Delta_k > 0$ means the current partial execution is preferable because it preserves accuracy while reducing latency; $\Delta_k < 0$ means additional agent interactions remain beneficial.

The controller is a small MLP that takes execution-state features as input and predicts $\hat{\Delta}_k$:

$$\hat{\Delta}_k = \text{Controller}_\phi \left(\frac{n_{\text{completed}}}{n_{\text{total}}}, \frac{\text{max_agree}}{n_{\text{completed}}}, \text{Compress}(q) \right), \quad (11)$$

where the first two features capture execution progress and answer consensus, and $\text{Compress}(q)$ is a learned low-dimensional projection of the query embedding for difficulty awareness. We define consensus via task-specific semantic equivalence of the current operator outputs: for GSM8K and MATH, we extract the boxed answer (or, if absent, the final numeric answer) and compare exact matches; for MMLU-Pro, we extract the predicted option letter and compare exact matches; for HumanEval, we regard two outputs as agreeing if they induce the same pass/fail outcomes on the test cases available during execution (i.e., the same execution feedback used by the **Test** operator). The predicted advantage is converted to a selection score via:

$$p(\text{select current}) = \sigma(\hat{\Delta}_k / \sigma_\Delta), \quad (12)$$

where $\sigma_\Delta = \text{std}(\{\Delta_k \mid \Delta_k > 0\})$ is computed from training data. At inference time, we sample the retain/eliminate decision from this Bernoulli score at each decision point.

Controller training. The execution controller is trained *after* the orchestrator converges, using execution traces collected by running the frozen orchestrator on the training set. For each query, all operators are executed to completion, and Δ_k is computed for every decision point k . The controller is trained via MSE regression: $\min_\phi \mathbb{E}[(\hat{\Delta}_k - \Delta_k)^2]$. Since R_k uses the same Lagrangian reward as the orchestrator, the controller inherits the orchestrator’s latency-aware objective under the same accuracy floor—in particular, the dual variable λ_S encodes how difficult accuracy is to maintain, automatically calibrating the controller’s aggressiveness.

The full training and inference procedure is summarized in Appendix C.5 (Algorithm 1). Phase 1 trains the orchestrator with Lagrangian-constrained policy gradient and critical-path credit assignment. Phase 2 trains the execution controller on traces from the frozen orchestrator to predict the control advantage.

4 Experiments

4.1 Experimental Setup

Benchmarks & Tasks. We evaluate our method on four benchmarks spanning three task categories. For HUMANEval, GSM8K, and MATH, we follow the same training and evaluation splits as MaAS Zhang et al. [2025a]; for MMLU-PRO Wang et al. [2024], we construct our own stratified train/test split from the official test set. Detailed split statistics are provided in Appendix B.3. For **code generation**, we use HUMANEval, which measures functional correctness of Python programs generated from natural language descriptions. For **mathematical reasoning**, we use GSM8K and MATH, which consist of grade-school and competition-level math problems requiring multi-step reasoning. For **knowledge reasoning**, we use MMLU-PRO, which covers diverse professional-level questions requiring multi-step reasoning.

Baselines. We compare against two categories of baselines: (i) **non-MAS prompting strategies**—**Generate** (direct single-step answer), **CoT** Wei et al. [2022], and **SC** Wang et al. [2022]; and (ii) **learned MAS orchestrators**—**GDesigner** Zhang et al. [2024], **AgentDropout** Wang et al. [2025c], and **MaAS** Zhang et al. [2025a]. Detailed per-baseline configurations are provided in Appendix B.4.

Evaluation Metrics. We evaluate all methods using three metrics: task performance, API cost, and latency. For task performance, we report pass@1 on HUMANEval, and accuracy on GSM8K, MATH, and MMLU-PRO. API cost is measured as the average per-query monetary cost (in units of $\times 10^{-4}$ USD) incurred by LLM API calls. For latency, we directly measure the end-to-end wall-clock execution time, reported as the average latency per query (in seconds). All methods in Tables 1–3 that use API-served backbones are evaluated under the same API provider and execution runtime, without response caching, and latency is averaged over 3 independent runs. The end-to-end latency is determined by the critical execution path as defined in Section 3.1. We also report CP Tokens, the average number of output tokens generated along the critical execution path, as a complementary structural indicator of critical-path workload.

Table 1: Comparison with non-MAS and MAS baselines across four benchmarks. Each cell reports mean $\pm$ standard deviation across 3 independent runs. For MAS methods, the best mean in each column per dataset is shown in **bold**. Cost is reported in units of $\times 10^{-4}$ USD. Latency is the average end-to-end wall-clock time per query (in seconds). CP Tokens denotes the average number of output tokens on the critical execution path.

Dataset	Method	Acc (%) $\uparrow$	Cost ($\times 10^{-4}$) $\downarrow$	Latency (s) $\downarrow$	CP Tokens $\downarrow$
GSM8K	Generate	87.46 $\pm$ 0.20	2.87 $\pm$ 0.05	5.43 $\pm$ 0.18	280 $\pm$ 8
	CoT	88.97 $\pm$ 0.43	4.53 $\pm$ 0.04	6.92 $\pm$ 0.21	331 $\pm$ 15
	SC	89.86 $\pm$ 0.19	18.61 $\pm$ 0.09	10.64 $\pm$ 0.27	591 $\pm$ 22
	GDesigner	93.30 $\pm$ 0.93	17.86 $\pm$ 0.28	32.31 $\pm$ 1.07	1624 $\pm$ 29
	AgentDropout	93.11 $\pm$ 0.27	53.81 $\pm$ 6.78	81.24 $\pm$ 12.74	4098 $\pm$ 538
	MaAS	93.36 $\pm$ 0.72	31.52 $\pm$ 0.78	48.11 $\pm$ 3.49	2501 $\pm$ 53
	LAMaS	93.65$\pm$0.41	7.71$\pm$0.94	11.73$\pm$1.13	622$\pm$87
HumanEval	Generate	84.73 $\pm$ 1.32	5.11 $\pm$ 0.06	10.23 $\pm$ 0.71	513 $\pm$ 31
	CoT	85.50 $\pm$ 3.05	7.24 $\pm$ 0.32	14.87 $\pm$ 1.04	780 $\pm$ 47
	SC	86.26 $\pm$ 0.76	28.15 $\pm$ 0.13	20.41 $\pm$ 0.57	1105 $\pm$ 38
	GDesigner	85.50 $\pm$ 2.02	19.86 $\pm$ 2.46	49.68 $\pm$ 5.62	2983 $\pm$ 489
	AgentDropout	86.01 $\pm$ 2.20	27.73 $\pm$ 1.32	51.49 $\pm$ 6.93	3043 $\pm$ 218
	MaAS	93.38 $\pm$ 1.17	28.18 $\pm$ 1.21	47.32 $\pm$ 2.93	2708 $\pm$ 109
	LAMaS	95.42$\pm$1.53	12.92$\pm$1.16	21.00$\pm$2.84	1180$\pm$187
MATH	Generate	50.62 $\pm$ 1.09	6.38 $\pm$ 0.14	10.17 $\pm$ 0.34	491 $\pm$ 19
	CoT	51.23 $\pm$ 1.15	10.24 $\pm$ 0.07	21.03 $\pm$ 0.31	992 $\pm$ 21
	SC	51.85 $\pm$ 1.23	42.41 $\pm$ 0.27	27.62 $\pm$ 0.41	1340 $\pm$ 24
	GDesigner	50.82 $\pm$ 0.71	30.17 $\pm$ 2.27	60.71 $\pm$ 5.83	3155 $\pm$ 246
	AgentDropout	52.47 $\pm$ 0.94	97.23 $\pm$ 8.61	176.42 $\pm$ 19.18	8693 $\pm$ 974
	MaAS	52.67 $\pm$ 0.82	65.39 $\pm$ 1.19	109.87 $\pm$ 5.92	5602 $\pm$ 116
	LAMaS	53.29$\pm$1.25	28.74$\pm$3.05	28.41$\pm$4.27	1452$\pm$232
MMLU-Pro	Generate	54.47 $\pm$ 1.14	1.05 $\pm$ 0.08	2.68 $\pm$ 0.22	136 $\pm$ 7
	CoT	56.80 $\pm$ 3.03	2.41 $\pm$ 0.15	6.27 $\pm$ 0.46	361 $\pm$ 14
	SC	58.67 $\pm$ 1.21	14.88 $\pm$ 0.18	10.53 $\pm$ 0.49	620 $\pm$ 12
	GDesigner	64.27 $\pm$ 3.63	25.03 $\pm$ 0.42	46.66 $\pm$ 2.47	2750 $\pm$ 41
	AgentDropout	61.87 $\pm$ 2.31	76.13 $\pm$ 1.41	119.33 $\pm$ 14.68	6560 $\pm$ 561
	MaAS	65.40 $\pm$ 3.08	30.37 $\pm$ 2.56	49.12 $\pm$ 5.13	2985 $\pm$ 151
	LAMaS	66.27$\pm$1.96	10.93$\pm$1.39	14.81$\pm$1.92	845$\pm$96

Implementation Details. We use the closed-source LLM gpt-4o-mini-0718 OpenAI [2024]. The latency penalty is set to $\lambda_t = 0.005$, the activation threshold to $\tau = 0.3$, and the supernet has $L = 4$ layers; we report sensitivity to each of these in the appendix. The target accuracy floor S_0 is set to the validation accuracy of the same orchestrator trained without latency optimization (i.e., $\lambda_t = 0$), so the constraint measures whether latency reduction is achieved while maintaining the reference accuracy level of the underlying architecture. Remaining training hyperparameters are listed in Appendix B.2. All results are averaged over 3 independent runs.

4.2 Result Analysis

We analyze the results from three perspectives: evaluated learning-based MAS baselines, the full set of baselines, and component-wise ablations.

Comparison with evaluated learning-based MAS baselines. Table 1 shows that LAMaS consistently reduces end-to-end latency by 55.6–75.6% relative to the strongest evaluated learning-based MAS baselines while maintaining competitive or better accuracy. The reduction co-occurs with substantially lower CP Tokens (e.g., 622 vs. 2501 on GSM8K), indicating that the orchestrator produces graphs with shorter critical paths under the latency objective; the inference-time controller further trims execution once a confident answer emerges.

Comparison with baselines. Non-MAS methods (**Generate**, **CoT**, **SC**) remain faster because of their simplicity but consistently underperform in task accuracy, especially on harder benchmarks.

Table 2: Ablation study of inference-time execution control, critical-path (CP) credit assignment, and the latency objective across four benchmarks. Cost is in units of $\times 10^{-4}$ USD; latency is end-to-end wall-clock time per query (in seconds).

Dataset	Method	Acc (%) $\uparrow$	Cost ($\times 10^{-4}$) $\downarrow$	Latency (s) $\downarrow$
GSM8K	LAMaS	93.65	7.71	11.73
	w/o controller	93.27	9.43	13.84
	w/o CP Credit	92.73	8.71	12.93
	w/o latency obj. ($\lambda_t=0$)	93.59	9.71	15.83
	naive lat. only	91.85	9.86	15.52
HumanEval	LAMaS	95.42	12.92	21.00
	w/o controller	95.17	16.18	24.47
	w/o CP Credit	94.40	14.83	23.43
	w/o latency obj. ($\lambda_t=0$)	95.67	17.42	35.62
	naive lat. only	93.13	18.07	31.84
MATH	LAMaS	53.29	28.74	28.41
	w/o controller	53.02	34.51	33.46
	w/o CP Credit	52.26	32.18	31.27
	w/o latency obj. ($\lambda_t=0$)	53.16	36.84	35.27
	naive lat. only	51.99	37.21	34.27
MMLU-Pro	LAMaS	66.27	10.93	14.81
	w/o controller	66.00	12.68	17.94
	w/o CP Credit	65.40	11.83	16.18
	w/o latency obj. ($\lambda_t=0$)	66.33	13.05	22.46
	naive lat. only	60.93	13.74	20.91

Among MAS methods, **GDesigner** and **AgentDropout** incur substantially higher latency without corresponding accuracy gains, whereas LAMaS achieves the lowest latency across all four benchmarks while maintaining comparable accuracy. Across MAS methods and benchmarks, CP Tokens are strongly correlated with measured wall-clock latency (Appendix C.1), providing complementary evidence that LAMaS reduces critical-path workload together with end-to-end latency.

4.3 Ablation Study

We study four variants of LAMaS: removing CP credit assignment, removing the inference-time controller, removing the latency objective, and a naive “add latency to the reward” baseline. *w/o CP Credit* forces the latency penalty to apply uniformly to all operators rather than weighting it by path criticality; *w/o controller* removes the execution controller entirely and relies solely on training-time optimization; *w/o latency obj.* sets $\lambda_t = 0$, training the orchestrator without any latency objective; *naive lat. only* keeps only a bare uniform latency term in the training objective, with no S_0 accuracy floor, no CP-aware credit assignment, and no inference controller—the simplest “add latency to the reward” baseline.

Table 2 shows consistent trends across all four benchmarks. Removing the controller preserves accuracy within 0.38 pp but increases latency by 16.5–21.1% and cost by 16.0–25.2%, confirming the importance of inference-time execution control. Removing CP credit assignment also degrades all three metrics, supporting path-criticality-weighted latency attribution rather than a uniform penalty over all operators. Removing the latency objective entirely ($\lambda_t=0$) leaves accuracy essentially unchanged (within ± 0.25 pp of LAMaS) but inflates latency by 24–70% and cost by 19–35%, confirming that the constrained latency objective reshapes the execution graph for efficiency while maintaining the reference accuracy level of the underlying architecture. Finally, the *naive lat. only* variant drops accuracy by 1.30–5.34 pp relative to LAMaS while yielding only 2–11% latency savings over $\lambda_t=0$ and no cost reduction, showing that simply adding a latency term provides only limited gains. Taken together, these ablations show that LAMaS combines path-selective latency pressure with an explicit accuracy floor and execution-time control rather than simply shrinking computation.

Table 3: Transferability of the full LAMaS pipeline (training-time graph optimization, critical-path credit assignment, and inference-time execution control) to other learning-based MAS architectures. The controller is retrained on each architecture’s own traces (Phase 2 of the LAMaS training procedure). Latency is the average end-to-end wall-clock time per query (in seconds). CP Tokens denotes the average critical-path output tokens.

Dataset	Method	Acc (%) $\uparrow$	Cost ($\times 10^{-4}$) $\downarrow$	Latency (s) $\downarrow$	CP Tokens $\downarrow$
GSM8K	GDesigner	93.30	17.86	32.31	1624
	GDesigner+LAMaS	93.27	12.08	20.55	1138
	AgentDropout	93.11	53.81	81.24	4098
	AgentDropout+LAMaS	93.18	39.67	58.14	2989
HumanEval	GDesigner	85.50	19.86	49.68	2983
	GDesigner+LAMaS	85.75	12.93	30.85	1571
	AgentDropout	86.01	27.73	51.49	3043
	AgentDropout+LAMaS	86.26	17.98	36.62	2200
MATH	GDesigner	50.82	30.17	60.71	3155
	GDesigner+LAMaS	51.23	20.03	39.93	2118
	AgentDropout	52.47	97.23	176.42	8693
	AgentDropout+LAMaS	52.40	70.35	128.34	6311
MMLU-Pro	GDesigner	64.27	25.03	46.66	2750
	GDesigner+LAMaS	64.33	17.03	28.27	1548
	AgentDropout	61.87	76.13	119.33	6560
	AgentDropout+LAMaS	62.20	56.23	82.27	4628

4.4 Transferability and Generality of LAMaS Mechanisms

The full LAMaS pipeline—the training-time graph optimization, the critical-path credit assignment, and the inference-time execution controller—is not tied to a specific orchestrator architecture. To verify its broader applicability, we integrate the entire pipeline into two other learning-based MAS methods, GDesigner Zhang et al. [2024] and AgentDropout Wang et al. [2025c], without redesigning their original orchestration frameworks.

Across all four benchmarks and both architectures, integrating the full LAMaS pipeline consistently reduces latency by 27–39% while preserving or slightly improving accuracy within typical 3-run noise. These results confirm that latency-aware graph optimization together with inference-time execution control transfers effectively to other learning-based MAS architectures. The reductions are smaller than in the LAMaS-native setting (Table 1), consistent with the fact that GDesigner and AgentDropout retain their original orchestration policies: the LAMaS pipeline contributes the latency-aware credit signal and inference-time control rather than replacing their graph construction. Together with Appendix E, this suggests that the LAMaS pipeline remains effective beyond a single orchestration architecture and extends to the additional backbone and serving configurations we evaluate. Appendix C.4 further shows that the orchestrator can still reasonably select an operator not exposed during training.

5 Conclusion

This paper addresses a key limitation of existing multi-agent orchestration methods: optimizing accuracy and cost alone does not reliably control execution latency. We propose LAMaS, which treats latency as an explicit optimization objective at two levels: training-time graph optimization with critical-path credit assignment, and inference-time adaptive execution control via a lightweight learned controller. Experiments across four benchmarks demonstrate that LAMaS achieves the best latency among evaluated learning-based MAS baselines, reducing end-to-end latency by over 50% while maintaining competitive or better accuracy, and the full pipeline transfers to other MAS architectures with consistent latency reductions.

References

- Reyna Abhyankar, Qi Qi, and Yiying Zhang. Osworld-human: Benchmarking the efficiency of computer-use agents. *arXiv preprint arXiv:2506.16042*, 2025.
- Jinyuan Chen, Jiuchen Shi, Quan Chen, and Minyi Guo. Kairos: Low-latency multi-agent serving with shared llms and excessive loads in the public cloud. *arXiv preprint arXiv:2508.06948*, 2025.
- Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.
- Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.
- Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021.
- Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, et al. Metagpt: Meta programming for a multi-agent collaborative framework. In *The twelfth international conference on learning representations*, 2023.
- Hao Kang, Qingru Zhang, Han Cai, Weiyuan Xu, Tushar Krishna, Yilun Du, and Tsachy Weissman. Win fast or lose slow: Balancing speed and accuracy in latency-sensitive decisions of llms. *arXiv preprint arXiv:2505.19481*, 2025.
- Guohao Li, Hasan Hammoud, Hani Itani, Dmitrii Khizbullin, and Bernard Ghanem. Camel: Communicative agents for "mind" exploration of large language model society. *Advances in neural information processing systems*, 36:51991–52008, 2023.
- OpenAI. GPT-4o mini: advancing cost-efficient intelligence. <https://openai.com/index/gpt-4o-mini-advancing-cost-efficient-intelligence/>, July 2024. Accessed: 2026-01-06.
- Song Wang, Zhen Tan, Zihan Chen, Shuang Zhou, Tianlong Chen, and Jundong Li. Anymac: Cascading flexible multi-agent collaboration via next-agent prediction. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 11566–11578, 2025a.
- Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.
- Yingxu Wang, Siwei Liu, Jinyuan Fang, and Zaiqiao Meng. Evoagentx: An automated framework for evolving agentic workflows. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 643–655, 2025b.
- Yubo Wang, Xueguang Ma, Ge Zhang, Yuansheng Ni, Abhranil Chandra, Shiguang Guo, Weiming Ren, Aaran Arulraj, Xuan He, Ziyang Jiang, et al. Mmlu-pro: A more robust and challenging multi-task language understanding benchmark. *Advances in Neural Information Processing Systems*, 37: 95266–95290, 2024.
- Zhexuan Wang, Yutong Wang, Xuebo Liu, Liang Ding, Miao Zhang, Jie Liu, and Min Zhang. Agentdropout: Dynamic agent elimination for token-efficient and high-performance llm-based multi-agent collaboration. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 24013–24035, 2025c.
- Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. Autogen: Enabling next-gen llm applications via multi-agent conversations. In *First conference on language modeling*, 2024.

Yu Yao, Yiliao Song, Yian Xie, Mengdan Fan, Mingyu Guo, and Tongliang Liu. Can dependencies induced by llm-agent workflows be trusted? In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

Guibin Zhang, Yanwei Yue, Xiangguo Sun, Guancheng Wan, Miao Yu, Junfeng Fang, Kun Wang, Tianlong Chen, and Dawei Cheng. G-designer: Architecting multi-agent communication topologies via graph neural networks. *arXiv preprint arXiv:2410.11782*, 2024.

Guibin Zhang, Luyang Niu, Junfeng Fang, Kun Wang, LEI BAI, and Xiang Wang. Multi-agent architecture search via agentic supernet. In *Forty-second International Conference on Machine Learning*, 2025a. URL <https://openreview.net/forum?id=imcyVlzpXh>.

Guibin Zhang, Yanwei Yue, Zhixun Li, Sukwon Yun, Guancheng Wan, Kun Wang, Dawei Cheng, Jeffrey Xu Yu, and Tianlong Chen. Cut the crap: An economical communication pipeline for LLM-based multi-agent systems. In *The Thirteenth International Conference on Learning Representations*, 2025b. URL <https://openreview.net/forum?id=LkzuPorQ5L>.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xiong-Hui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating agentic workflow generation. In *The Thirteenth International Conference on Learning Representations*, 2025c. URL <https://openreview.net/forum?id=z5uVAKwmjf>.

A Limitations

LAMaS reduces orchestration latency; however, it does not mitigate broader risks inherited from the underlying LLMs and associated tools, including unsafe outputs, hallucinations, and social biases.

B Experimental Setup Details

This section collects additional implementation and dataset details used for reproducibility.

B.1 Operator Set

Each operator corresponds to a predefined reasoning or execution primitive used to construct execution graphs. We briefly summarize their functionality below for completeness.

- **Generate.** A basic generator that directly produces text or code without additional reasoning or post-processing. It invokes the LLM once and is primarily used for simple generation tasks.
- **GenerateCoT.** A chain-of-thought generator that prompts the LLM to perform step-by-step reasoning. For mathematical tasks (e.g., MATH, GSM8K), it includes explicit reasoning exemplars, while for code tasks (e.g., HumanEval), it uses lightweight reasoning prompts. This operator invokes the LLM once.
- **MultiGenerateCoT.** A diversity-oriented CoT generator that produces multiple candidate solutions in parallel. It generates three independent chain-of-thought solutions, yielding a set of candidate responses for downstream aggregation.
- **ScEnsemble.** A self-consistency ensemble operator that selects the most consistent answer from multiple candidate solutions. All candidates are formatted as discrete options, and the LLM is prompted to select the most consistent one, following the self-consistency principle.
- **SelfRefine.** A refinement operator that analyzes an existing solution to identify errors or suboptimal reasoning and generates an improved version. This operator invokes the LLM once.

Task-Specific Operators. We also employ task-specific operators for different benchmarks.

- **CustomCodeGenerate** (HumanEval). A lightweight code generator that produces candidate code solutions without execution or testing.
- **Test** (HumanEval). A test-driven refinement operator that executes generated code and iteratively improves it based on failure feedback. Upon failure, the operator generates a revised solution using reflective prompts and retries up to three times.
- **Programmer** (MATH/GSM8K). A code execution operator that generates Python programs, executes them in an isolated environment, and iteratively refines the code based on execution feedback.

B.2 Additional Implementation Details

We list the remaining training hyperparameters not detailed in Section 4.1. The cost weight is $\lambda_c = 3.0$, the dual learning rate is $\eta = 0.01$, the batch size is 4, and the orchestrator is trained for 4 repetitions over the training set. These values follow the supernet implementation conventions of MaAS Zhang et al. [2025a].

Queries are encoded by a frozen `all-MiniLM-L6-v2` SentenceTransformer; the resulting embedding is shared between the orchestrator’s policy network (Section 3.2) and the controller’s `Compress(·)` projection (Section 3.6), and is not finetuned during training.

In the main experiments, we use `gpt-4o-mini-0718` via the OpenAI API with temperature set to 1. Appendix E instead evaluates `qwen3.5-flash` (closed-source) via OpenRouter with Alibaba Cloud `Int.` selected as the provider, and `qwen3.5-9b` (open-weight) served locally through vLLM; the same temperature setting is used there as well.

Training the orchestrator and execution controller requires modest local compute but does not finetune the underlying LLM backbone. All training runs were executed on a Slurm cluster using a single NVIDIA H100 80GB GPU per run. The local `qwen3.5-9b` evaluation uses vLLM served on two NVIDIA B200 GPUs.

B.3 Dataset Split Statistics

We evaluate on four publicly available benchmarks: HUMANEval Chen et al. [2021], GSM8K Cobbe et al. [2021], MATH Hendrycks et al. [2021], and MMLU-PRO Wang et al. [2024]. Following common practice in workflow-automation work [Zhang et al., 2025a,c], each benchmark is partitioned into training and test sets with a train:test ratio of 1:4. For MATH, we use a 605-problem subset of level-5 problems spanning four representative categories: Combinatorics & Probability, Number Theory, Pre-algebra, and Pre-calculus. For MMLU-PRO, we construct a stratified split from the official test set (seed 42), sampling proportionally per category with no overlap between train and test. Statistics are summarized in Table 4.

Table 4: Dataset statistics.

Domain	Dataset	#Train	#Test	Metric
Code Generation	HumanEval	33	131	pass@1
Math Reasoning	GSM8K	264	1055	Accuracy
	MATH	119	486	Accuracy
Knowledge Reasoning	MMLU-Pro	125	500	Accuracy

B.4 Baseline Setups

We provide the configurations for each baseline method below.

1. **Generate.** A single-pass direct-answer baseline that prompts the LLM once with the query and uses its output as the final answer, with no chain-of-thought scaffolding or post-processing.
2. **CoT.** Chain-of-Thought prompting Wei et al. [2022] elicits step-by-step reasoning before the final answer; we use a standard zero-shot CoT prompt template.
3. **SC.** Self-Consistency Wang et al. [2022]: to enhance robustness, we aggregate five CoT-generated solutions through an ensemble step over the candidate answers.

4. **GDesigner.** We use the official implementation of Zhang et al. [2024], which learns a graph neural network over operator embeddings to construct communication topologies. We retain the original training procedure and operator set defined in the released codebase, replacing only the LLM backbone with gpt-4o-mini-0718 for fair comparison.
5. **AgentDropout.** We follow the released implementation of Wang et al. [2025c], which dynamically prunes redundant agents and communication edges during execution. The dropout schedule and per-agent role assignments use the defaults from the official codebase.
6. **MaAS.** We use the agentic-supernet implementation of Zhang et al. [2025a], training the orchestrator with the same supernet depth ($L = 4$), operator set, and policy-gradient procedure as released. The original cost-only reward is used unchanged so that MaAS represents the cost-aware—but not latency-aware—learning baseline.

All MAS baselines are trained and evaluated under the same train/test partition (Appendix B.3), the same backbone LLM (gpt-4o-mini-0718), and the same per-query latency/cost measurement protocol described in Section 4.1.

C Additional Supporting Analyses

This section provides supplementary analyses supporting the main-paper interpretation of LAMaS.

C.1 Critical-Path Tokens as a Latency Proxy

The critical-path-aware credit assignment in Section 3.4 is motivated by the expectation that CP Tokens should co-vary with end-to-end wall-clock latency. To verify this empirically, we form one (CP Tokens, latency) pair for each MAS-method/benchmark cell in Table 1 and compute the pooled correlation across all 4 MAS methods $\times$ 4 benchmarks ($n = 16$). Both Pearson and Spearman correlations indicate a strong positive association, providing complementary evidence that reductions in CP Tokens align with reductions in end-to-end latency.

Table 5: Correlation between CP Tokens and measured wall-clock latency across MAS-method/benchmark cells in Table 1.

Setting	Pearson r	Spearman ρ
MAS methods $\times$ 4 benchmarks	0.901	0.878

C.2 Comparison with Oracle Controller

To probe how close the learned execution controller is to an optimal stopping rule, we compare it against an oracle that has access to the true control advantage $\Delta_k = R_k - R_N$ at each decision point and stops iff $\Delta_k > 0$. We evaluate on HumanEval and MMLU-Pro from two complementary angles: decision-level agreement with the oracle (Table 6), and end-to-end performance when the learned controller is replaced by either the oracle or no controller (Table 7).

Decision-level agreement. Table 6 reports decision accuracy together with precision, recall, and F1 for the positive (stop) class, treating the oracle’s $\Delta_k > 0$ rule as ground truth. Across both benchmarks, the learned controller agrees with oracle decisions in 70.8–73.5% of cases. Precision exceeds recall on both benchmarks, indicating that the learned controller is conservative: it tends to continue execution when the oracle’s signal is borderline rather than making an over-aggressive elimination decision.

Table 6: Decision-level agreement between the learned execution controller and the oracle stopping rule on HumanEval and MMLU-Pro. The positive class is the “stop” decision.

Benchmark	Decision Acc (%) $\uparrow$	Precision $\uparrow$	Recall $\uparrow$	F1 $\uparrow$
HumanEval	73.5	0.72	0.66	0.69
MMLU-Pro	70.8	0.69	0.63	0.66

End-to-end performance. Table 7 compares three controller settings: no controller, the learned controller, and the oracle. The learned controller closes most of the gap between no-controller

execution and the oracle: on HumanEval it trails the oracle by 2.16 s in latency and 0.25 pp in accuracy, and on MMLU-Pro by 1.74 s and 0.46 pp. This indicates that the learned controller approaches oracle-level execution control despite operating only on the partial-execution features available at runtime.

Table 7: End-to-end accuracy and latency under different controllers on HumanEval and MMLU-Pro. The “Learned” rows correspond to the LAMaS configuration in Table 2, and “No controller” rows correspond to the *w/o controller* ablation.

Benchmark	Controller	Acc (%) $\uparrow$	Latency (s) $\downarrow$
HumanEval	No controller	95.17	24.47
	Learned	95.42	21.00
	Oracle	95.67	18.84
MMLU-Pro	No controller	66.00	17.94
	Learned	66.27	14.81
	Oracle	66.73	13.07

C.3 Training Dynamics

Figure 3 shows the per-step batch-averaged accuracy $\bar{S}_t$, end-to-end latency $\bar{T}_t$, and dual variable λ_S over training, on GSM8K (264 optimizer steps) and MMLU-Pro (125 steps). Curves are smoothed with a trailing window of size 5 (size 3 for λ_S , to preserve its responsiveness to per-step accuracy fluctuations).

Three trends are visible. (i) Latency decays monotonically from a high initial value to the LAMaS configuration reported in Table 2, confirming that latency is the actively optimized quantity. (ii) Accuracy oscillates around the floor S_0 without a clear trend, since S_0 is a constraint rather than an objective. (iii) The dual variable λ_S performs a small random walk near 1.0 in response to the per-step violation $1 - \bar{S}_t/S_0$. On the harder MMLU-Pro task, λ_S shows a more pronounced early rise (the constraint is tighter when accuracy is harder to maintain), then partially relaxes; on the easier GSM8K task, λ_S stays close to 1.0 throughout. These dynamics match the intended behavior of the dual update in Eq. 6.

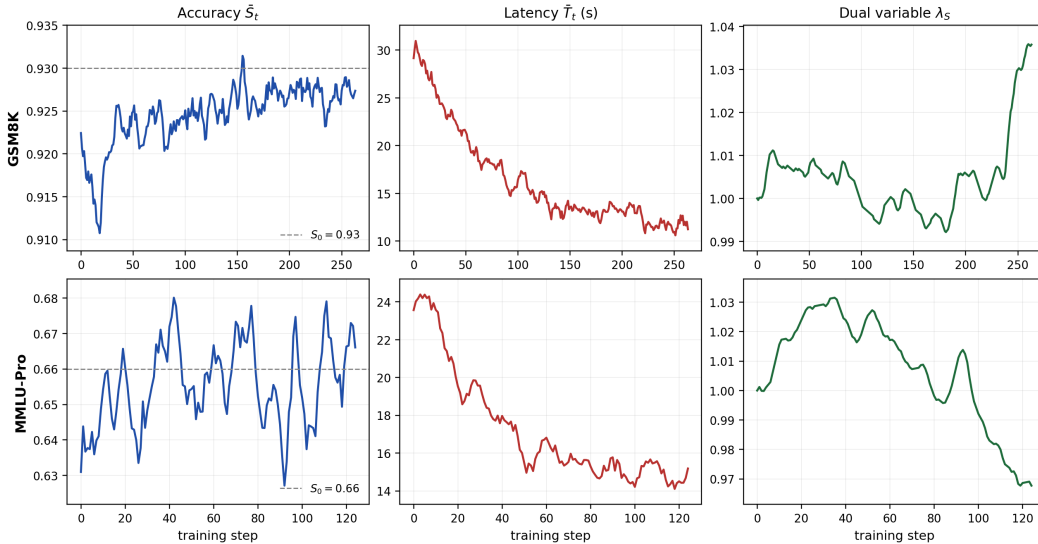


Figure 3: Training dynamics on GSM8K (top, 264 steps) and MMLU-Pro (bottom, 125 steps). Left: batch-averaged accuracy $\bar{S}_t$ with the accuracy floor S_0 marked by the dashed line. Middle: end-to-end latency $\bar{T}_t$ (s). Right: dual variable λ_S updated by Eq. 6.

C.4 Compatibility with Unseen Operators

To probe whether the learned policy generalizes beyond the fixed training-time operator set, we insert an unseen operator (**Debate**) at inference time on HumanEval and inspect the orchestrator’s average selection probabilities. Figure 4 reports the average probability mass over the seven non-halt operators shown. The unseen **Debate** operator receives 14.8% probability mass, comparable to seen operators such as **Generate** (14.9%), **ScEnsemble** (15.1%), and **Test** (15.5%). This suggests that the learned policy can still reasonably select an unseen operator during architecture sampling, indicating some inductive flexibility beyond the exact operator set seen during training.

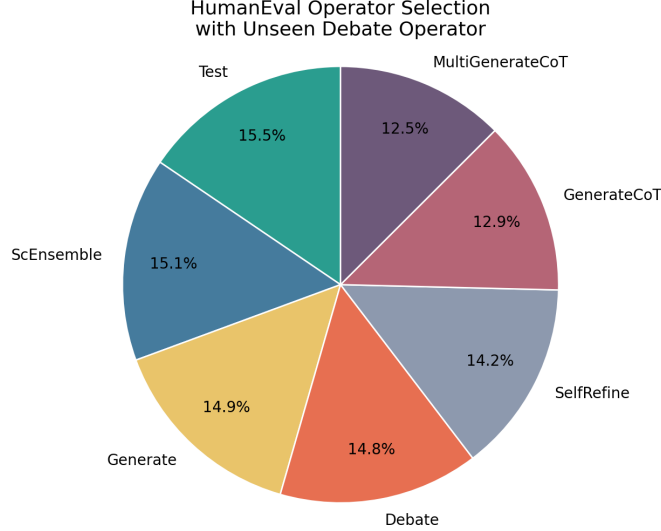


Figure 4: Average operator selection probabilities on HumanEval after inserting an unseen operator, **Debate**, at inference time. The unseen operator receives non-trivial probability mass comparable to several seen operators, indicating that the learned policy can still reasonably select operators not exposed during training.

C.5 Full Training and Inference Procedure

D Sensitivity Analyses

This section reports the sensitivity of LAMaS to the main hyperparameters that control latency pressure, graph structure, and the accuracy floor.

D.1 Parameter Sensitivity Analysis

We analyze the sensitivity of LAMaS to four key hyperparameters: the latency penalty weight λ_t , the activation threshold τ , the number of supernet layers L , and the accuracy floor S_0 . We report results on HumanEval and MMLU-Pro, with the default configuration ($\lambda_t=0.005$, $\tau=0.3$, $L=4$) marked by $\blacklozenge$.

Latency weight λ_t . Table 8 shows that increasing λ_t monotonically reduces latency, while accuracy varies only mildly across the sweep. On HumanEval, raising λ_t from 0 to 0.05 reduces latency from 35.62s to 11.29s with at most 2.0% accuracy change; the trend is consistent on MMLU-Pro. The $\lambda_t=0$ setting (no latency objective during training) leaves accuracy near the LAMaS level but yields the largest latency, complementing the corresponding row in the ablation study (Table 2). Overall, λ_t provides smooth and predictable control over latency while keeping accuracy broadly stable.

Activation threshold τ . Table 9 reveals that the threshold controls the width of the execution graph. A lower threshold ($\tau=0.1$) produces lean graphs with fewer operators per step, yielding the lowest latency but reduced accuracy. A higher threshold ($\tau=0.4$) activates more operators, improving

Algorithm 1 LAMaS: Latency-Aware MAS Orchestration

Require: Orchestrator π_θ , operator set $\mathcal{O}$, training queries $\mathcal{D}$, accuracy floor S_0

Require: Hyperparameters: λ_t, λ_c , dual learning rate η , threshold τ

```
1: // Phase 1: Train orchestrator
2: Initialize dual variable  $\lambda_S \leftarrow 1.0$ 
3: for each training iteration do
4:   Sample a batch of queries  $\{x_i\}$  from  $\mathcal{D}$ 
5:   for each query  $x_i$  do
6:     Construct execution graph  $\mathcal{G}_i$  via  $\pi_\theta$ 
7:     Execute  $\mathcal{G}_i$ ; observe  $S_i, C_i, T_i$ , per-operator times  $\{t(o)\}$ 
8:     Compute  $\ell(o)$  for all operators; compute  $R(o)$  via Eq. 8
9:   end for
10:  Update  $\theta$  via policy gradient (Eq. 9)
11:   $\lambda_S \leftarrow \max(0, \lambda_S + \eta(1 - \bar{S}/S_0))$ 
12: end for
13: // Phase 2: Train execution controller
14: Freeze  $\pi_\theta$ ; collect full execution traces on  $\mathcal{D}$ 
15: for each trace do
16:   for each decision point  $k = 1, \dots, N$  do
17:     Compute  $\Delta_k = R_k - R_N$  (Eq. 10)
18:     Record training sample:  $(f_k, \Delta_k)$ 
19:   end for
20: end for
21: Train Controller $_\phi$  via MSE:  $\min_\phi \sum (\hat{\Delta}_k - \Delta_k)^2$ 
22: Compute  $\sigma_\Delta = \text{std}(\{\Delta_k \mid \Delta_k > 0\})$ 
23: // Inference
24: Construct  $\mathcal{G}$  via  $\pi_\theta$ ; launch operators
25: for each operator completion  $k$  do
26:   Compute  $p(\text{select current})$  from Controller $_\phi(f_k)$ , sample a Bernoulli decision; on a positive draw, eliminate the following unlaunched operators.
27: end for
28: return Final answer
```

Table 8: Sensitivity to the latency penalty weight λ_t . Increasing λ_t consistently reduces latency, with mild accuracy variation across the sweep.

Dataset	λ_t	Acc (%) $\uparrow$	Latency (s) $\downarrow$
HumanEval	0	95.67	35.62
	0.001	95.93	23.50
	0.005 $\blacklozenge$	95.42	21.00
	0.01	94.91	13.91
	0.05	93.89	11.29
MMLU-Pro	0	66.33	22.46
	0.001	67.13	16.41
	0.005 $\blacklozenge$	66.27	14.81
	0.01	66.13	14.11
	0.05	65.27	11.38

accuracy through greater exploration together with increased latency. The default $\tau=0.3$ provides a balanced setting across both benchmarks.

Number of layers L . Table 10 shows that increasing the number of supernet layers provides greater reasoning depth. On HumanEval, accuracy improves from 94.15% ($L=2$) to 95.93% ($L=8$), but latency increases from 12.85s to 25.54s due to the deeper critical path. The marginal accuracy gain diminishes beyond $L=4$, while latency continues to grow, justifying the default choice of $L=4$.

Accuracy floor S_0 . Unlike λ_t, τ , and L , which control the latency penalty and graph structure, the accuracy floor S_0 sets the target accuracy level used during training. Table 11 reports results across $S_0 \in \{0.95, 1.00, 1.02, 1.05\} \cdot S_{\text{ref}}$, where S_{ref} denotes the orchestrator’s reference accuracy and the default deployment ($S_0 = S_{\text{ref}}$) is marked by $\blacklozenge$. Tightening S_0 towards S_{ref} enforces a stricter no-regression guarantee at the cost of more execution, while relaxing S_0 allows LAMaS to push more aggressively for latency reduction with a small accuracy concession. The trend is monotonic on

Table 9: Sensitivity to the activation threshold τ . A higher threshold activates more operators per step, increasing accuracy and latency. A lower threshold produces leaner graphs with lower latency but reduced accuracy.

Dataset	τ	Acc (%) $\uparrow$	Latency (s) $\downarrow$
HumanEval	0.1	93.89	13.57
	0.2	94.91	15.35
	0.3 $\blacklozenge$	95.42	21.00
	0.4	95.67	22.80
MMLU-Pro	0.1	65.13	13.42
	0.2	65.93	14.23
	0.3 $\blacklozenge$	66.27	14.81
	0.4	66.67	15.62

Table 10: Sensitivity to the number of supernet layers L . More layers provide greater reasoning depth and slightly higher accuracy, but increase latency due to deeper critical paths.

Dataset	L	Acc (%) $\uparrow$	Latency (s) $\downarrow$
HumanEval	2	94.15	12.85
	4 $\blacklozenge$	95.42	21.00
	6	95.67	23.50
	8	95.93	25.54
MMLU-Pro	2	64.93	12.81
	4 $\blacklozenge$	66.27	14.81
	6	66.47	16.56
	8	66.53	17.43

both benchmarks: on HumanEval, sweeping S_0 from $0.95 \cdot S_{\text{ref}}$ to $1.05 \cdot S_{\text{ref}}$ moves accuracy from 94.66% to 96.18% while latency rises from 17.85s to 23.54s; the same direction holds on MMLU-Pro (65.40% $\rightarrow$ 67.13%, 12.60s $\rightarrow$ 16.51s). This confirms that varying S_0 during training provides a smooth way to control how aggressively latency is reduced while maintaining different target accuracy floors.

Table 11: Sensitivity to the accuracy floor S_0 . We vary S_0 as a fraction of the architecture’s reference accuracy S_{ref} ($S_0 = S_{\text{ref}}$ is the default, marked by $\blacklozenge$).

Dataset	S_0/S_{ref}	Acc (%) $\uparrow$	Latency (s) $\downarrow$
HumanEval	0.95	94.66	17.85
	1.00 $\blacklozenge$	95.42	21.00
	1.02	95.67	22.00
	1.05	96.18	23.54
MMLU-Pro	0.95	65.40	12.60
	1.00 $\blacklozenge$	66.27	14.81
	1.02	66.67	15.46
	1.05	67.13	16.51

E Generalizability Across Backbones and Serving Setups

To assess whether LAMaS’s latency advantage transfers across different LLM backbones and serving setups, we replicate the comparison on two additional configurations: the closed-source flash variant qwen3.5-flash served through OpenRouter, and the open-weight qwen3.5-9b served locally with vLLM. Both are evaluated on HumanEval and MMLU-Pro against the strongest learning-based MAS baseline (MaAS) and LAMaS. As shown in Table 12, LAMaS reduces latency by about 60% on HumanEval and about 43% on MMLU-Pro across the two settings, while matching or slightly exceeding MaAS in accuracy. This confirms that LAMaS’s mechanisms transfer across distinct backbones and serving setups.

Table 12: Generalizability across Qwen backbones and serving setups: closed-source qwen3.5-flash served through OpenRouter and open-weight qwen3.5-9b served locally with vLLM. Latency is end-to-end wall-clock time per query (s). Best mean among MAS methods per column per dataset is in **bold**.

Dataset	Backbone	Method	Acc (%) $\uparrow$	Latency (s) $\downarrow$
HumanEval	qwen3.5-flash	MaAS	94.91	100.87
		LAMaS	96.95	38.93
	qwen3.5-9b	MaAS	95.17	105.18
		LAMaS	95.93	42.18
MMLU-Pro	qwen3.5-flash	MaAS	82.40	174.60
		LAMaS	82.93	98.04
	qwen3.5-9b	MaAS	71.20	178.62
		LAMaS	72.93	102.07